Friedrich-Alexander-Universität Erlangen-Nürnberg

Lehrstuhl für Multimediakommunikation und Signalverarbeitung

Prof. Dr.-Ing. André Kaup

Studienarbeit

Parallel Design of Generic Arithmetic Coding for the CITK

of Johannes Rehm

Oktober 2012

Supervisor: Andreas Weinlich

Erklärung

Ich versichere, dass ich die vorliegende Arbeit ohne fremde Hilfe und ohne Benutzung anderer als der angegebenen Quellen angefertigt habe, und dass die Arbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegen hat und von dieser als Teil einer Prüfungsleistung angenommen wurde. Alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, sind als solche gekennzeichnet.

_____ _____

Ort, Datum Unterschrift

Contents

Kurzfassung

Ein Framework für eine parallele arithmetische Codierung wird präsentiert, für das eine binäre und eine M-stufige Coding-Engine implementiert wurde. Hierfür wird die CUDA Architektur verwendet um den hoch parallelen Manycore-Prozessor eines Grafikprozessors (GPU) ausnützen zu können. Diese Arbeit konzentriert sich auf das verarbeiten von großen Datenmengen in kurzer Zeit mit einer hohen Codiereffizienz um z.B. 3-D Volumenbilder und -videos aus medizinschen Geräten komprimieren zu können. Dafür verwendet die Implementierung das Insight Segmentation and Registration Toolkit und die zugehörige Erweiterung, das CUDA Insight Toolkit.

Es werden unabhängige 2-D Blöcke parallel verarbeitet und dazu werden Techniken präsentiert um den zusätzliche Overhead aufgrund der Blockaufteilung zu minimieren. Der Overhead und der Verlust an Codiereffizienz wird mit einer sequenziellen Ausführung verglichen. Hierzu wurde eine gleichartige Implementierung, die auf der CPU läuft, erstellt und der Context-based Adaptive Binary Arithmetic Coding (CABAC) aus dem H.264/AVC Standard auf das ITK portiert.

In den Testergebnissen zeigt sich, dass die Verluste an Codiereffizienz der parallelen im Vergleich zu sequentiellen Ausführung sehr gering sind und eine wesentlich höhere Geschwindigkeit durch die parallel Verarbeitung von unabhängigen Blöcken auf der GPU erzielt werden kann.

Abstract

A parallel arithmetic coding framework is presented. A binary and a M-ary arithmetic coding engine is implemented using the CUDA architecture to capitalize on the highly parallel, many core processor of a graphics processing unit (GPU). This work concentrates on processing huge amount of data in a short time with a high coding efficiency to compress e.g. 3-D volume images and videos of medical modalities. Therefore, the implementation is build on the Insight Segmentation and Registration Toolkit and its extension the CUDA Insight Toolkit.

Unique 2-D coding blocks are processed in parallel and techniques are presented to minimize the additional overhead resulting from block partition. This overhead and its lost in coding efficiency is compared to sequential processing and therefore a similar implementation running on the the CPU is designed and the context-based adaptive binary arithmetic coder (CABAC) known from the H.264/AVC standard is ported to the ITK. As a result of the comparison it can be shown that the losses in coding efficiency are very small and a extensive speed up is achieved by the parallel processing of unique blocks running on a GPU.

List of Abbreviations

BAC	Binary Arithmetic Coding
CABAC	Context-based Adaptive Binary Arithmetic Coding
CITK	Cuda Insight Toolkit
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
GPU	Graphics Processing Unit
ITK	Insight Segmentation and Registration Toolkit
LPS	Least Probable Symbol
MAC	M-ary Arithmetic Coding
MPS	Most Probable Symbol
PBAC	Parallel Binary Arithmetic Coding
PMAC	Parallel M-ary Arithmetic Coding
SIMD	Single Instruction, Multiple Data
UEGk	Unary / kth order Exp-Golomb

Chapter 1

Introduction

The performance of computing hardware increases very fast after its invention. A very popular description of this circumstance is given by Moore's Law which claims that the number of transistors on integrated circuits doubles in a certain period of time (about two years or 18 month depending on the definition) [1]. Thus it is possible to compute increasingly complex algorithms at shorter periods of time.

Frequency scaling was the main reason for the improvements in performance for a very long time. But higher frequencies result in higher power consumptions and consequently in an increasing evolution of heat which constrains this scaling approach. Modern processors achieve higher throughput due to parallelization in which many calculations are carried out simultaneously [2].

To operate at full capacity, the algorithm or the data has to be partitioned and have to be computed independently. Traditional algorithms are designed for a sequential execution and a considerable parallelization is not possible in many cases.

For that reasons, new parallel procedures can solve the same issues as sequential procedures much faster on modern architectures.

Fast computations are needed for many tasks. One of them is image and video coding. This work is about a parallel design of arithmetic coding, an entropy coding mode which is used in most modern image and video codecs because it is a very efficient, near optimal, mode and an easy adaption on changing probability distributions within

the input signal is possible. The coding of medical volume images and videos does have a very high computational effort due to the large amount of data. But high amount of data leads to the possibility of a broad partition in this case.
The partition of data causes some loss in coding efficiency. For that reason a non-parallel implementation is also designed and compared regarding coding efficiency and execution time. This implementation runs on a common central processing unit (CPU). The highly parallel structure of a graphics processing unit (GPU) is used for the parallel implementation. The CUDA architecture offers a well documented and reliable base for this task, an overview about this architecture is given in an own paragraph.
The ITK is the software framework used in this work for both implementations. The CITK extend this framework by some support for CUDA. ITK and CITK are also explained in an own paragraph.
After a theoretical and a practical examination of the common arithmetic coding procedure, a widely used approach, the CABAC framework, known form the H.264/AVC and the HEVC standard, is introduced. Thereafter, the design of the binary arithmetic coder, elaborated in this work, is explained and at the end some results are given and explained.
A framework based on the CITK was implemented to examine all this tasks. The parallel arithmetic coding process including a standard M-ary arithmetic coder, based on [3] and a binary arithmetic coder were designed. The binary coder is very similar to the CABAC approach. Both are running on the GPU in parallel and non-parallel on the CPU. M-ary is related to the alphabet size of the input symbols which is 2^M in this case. A CPU implementation of the binary arithmetic coding part of the CABAC framework was also ported to this framework to make a reasonable comparison.

Chapter 2

Nvidia CUDA

Compute Unified Device Architecture (CUDA) is a general purpose parallel computing architecture developed by the Nvidia Corporation[4]. This architecture is introduced in the following sections and it is explained what has to be taken into account to achieve a high throughput.

2.1 SIMD Architecture

CUDA is running on Nvidia GPUs which consist of several multiprocessors which themself consist of a certain number of CUDA Cores, depending on the compute capability. The multiprocessors operate up to the principle of SIMD (= single instruction, multiple data)[5]. This means every operation in the program flow is executed by all CUDA Cores of a multiprocessor in parallel. Processing only differs by the input data. The advantage of this parallelism is, few transistors are needed for the control unit (as can be seen in Fig 2.1).

This architecture is exemplarily shown on the right part of Fig 2.1. Every line illustrates one multiprocessor and every green cell represents a CUDA Core. Every multiprocessor has its own control unit and on-board memory shown in yellow and orange color.

The warp scheduler creates, manages, schedules, and executes threads in groups of parallel threads, called warps. If, e. g., a warp is waiting for data, the warp scheduler

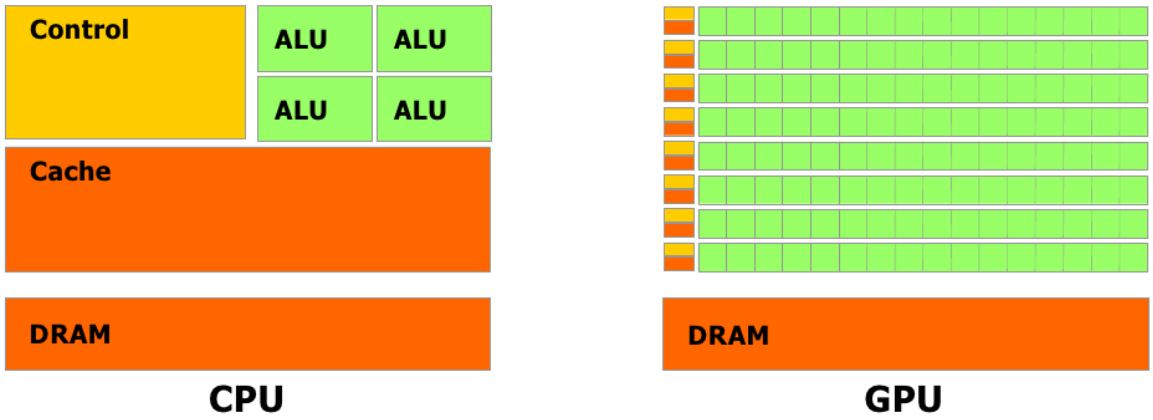


Figure 2.1: Outlined illustrations of the architectures of a CPU and a GPU[4]

can give compute capacity to another warp. Several warps should run on a multiprocessor to utilize the capacity. A GeForce 480 [6], e. g., consists of 15 multiprocessors with compute capability 2.0, thus 32 CUDA Cores, in total 480 CUDA Cores. About 16 warps ($16 \cdot 32 = 512$ Threads) per multiprocessor are required to operate at full capacity (this highly depends on the number and kind of memory accesses). So the great amount of 512 threads $\cdot 15$ multiprocessors $= 7680$ parallel threads is needed by this calculation to achieve the best throughput.

One big disadvantage of this architecture which has to be taken into account concerns the path divergence. If the execution paths diverge (e. g. loops with different number of iterations, or if-conditions depending on the input data), every thread has to wait for the ”slowest” thread of the warp. This circumstance does have a strong impact on the effective instruction throughput.

2.2 Memory Hierarchy

Today’s GPUs are very fast in computation and therefore most transistors are used for the arithmetic and logic units. Only few transistors are used for the control units as mentioned in the paragraph before and also the size of data caches, the on-chip memory, is very low in comparison to a CPU.

Memory which is used by CUDA programs is available at two different areas of the graphics card. The on-chip memory which is located directly at the GPU die and the device memory (DRAM in 2.1), a dynamic random-access memory plugged on a distinct chip.

Memory-intensive tasks are costly. Only a few clock-cycles are necessary to catch data from the on-chip memory. But it takes hundreds of clock-cycles till data from the device memory is available.

Shared memory is one way to use GPU-caching in an efficient way. This kind of fast onboard-memory can be allocated directly by the user and frequently accessed parts of the device memory or intermediate data can be saved there.

The CUDA architecture make it also possible to use textures of the graphics card. One important feature of texture memory is the corresponding cache. Texture cache is optimized for 2-D spatial locality and can speed up memory accesses on multidimensional data. Textures can be accessed by so called references. Texture references allow only readable access, but they do have other advantages which are described in detail in [4] and are not used in this work. Surface references, in contrast, allow writable access, but do not have the special features of texture references.

2.3 Instruction Throughput

The instruction throughput also differs from GPU to CPU. GPUs are very fast in 32-bit arithmetic operations as you can see in Table 2.1. Multiplication with floating-point values are twice as fast as integer multiplications on the GPU and take only one clock cycle or in other words, one multiprocessor can perform 32 floating-point multiplies at compute capability 2.0. Thus the differences to hardware implementations are very big as multiplies and divisions are very costly and bit operations are easy to implement.

Instructions	**GPU**	**CPU**
32-bit floating-point add, multiply	1	4
32-bit floating-point multiply-add	1	n.a.
32-bit integer add	1	1
32-bit integer compare	1	1
32-bit integer shift	2	1
32-bit integer multiply	2	4
32-bit integer multiply-add	2	n.a.

Table 2.1: Clock cycles for different arithmetic / logic operations by one CUDA Core with compute capability 2.0.[4] by comparison with a CPU[7]

Chapter 3

ITK: Insight Toolkit

The Insight Segmentation and Registration Toolkit (ITK) is an open-source software framework for medical image processing, segmentation and registration. The object-oriented language C++ is used for the implementation. It is maintained by the National Library of Medicine and developed by programmers from all over the world.
Typically the sampled representation, used for processing, is an image acquired from such medical modalities as CT or MRI scanners.

3.1 Data Processing Pipeline

The data processing pipeline is the basic method for data handling in the ITK. It consist of two basic elements. One or more data objects to represent the data and process objects to process the data. Process objects are again classed in three distinguish types. Sources provide data, filter objects gets as input and output data objects and mappers put out data. The pipeline has an automatic update mechanism, if the internal state or input of a process object changes it will be executed. [8]

3.2 CITK: Cuda Insight Toolkit

CITK is an extension to the present Insight Toolkit to support the CUDA parallel computing platform [9]. This extension manages image data on the host and on the device. The CudaImportImageContainer perform a copying if access to image data on the device is requested and if the current image data is available only on the host. This copying is also done the other way round if the data on the device is more up to date. But if the data is processed only on the GPU while consecutive filters are executed, the image data remains on the device and no copying is performed.

Chapter 4

Arithmetic Coding

4.1 Introduction

Arithmetic Coding is a very efficient Entropy Coding Mode. Entropy in case of information theory is a measure of the average information content in each symbol from a certain source [10]. The goal of Entropy Coding is to represent a message consisting of statistically independent symbols of a given alphabet with as few bits as possible in a reversible way. The entropy of a message, as it is defined by Claude Shannon, indicates a lower limit of needed bits to store the information. Arithmetic coding is used in most modern image and video coding ctandards like JPEG 2000 or H.264/AVC.

Another popular entropy coding mode is Huffman coding where each Symbol is represented by an integer number of bits. The number of bits is dependent on the probability of the symbol.

Arithmetic coding does not map every symbol to a codeword with a variable number of bits. It codes the whole message at once and make it possible to encode symbols using fragments of bits [3]. This characteristic is very important for high peaked probability distributions where the entropy of one or some symbols is much less than 1 Bit/Symbol. Huffman coding needs at least one bit per symbol.

A very important property of arithmetic coding is to have a clean interface between the

estimation of the symbol probabilities and the coding. The adaptation of the symbol probabilities do not lead to additional costs at the coding stage [11].

4.2 Encoding Procedure

Two elements are required for the arithmetic coding procedure, as you can read in the last section. An estimation of the probability distribution of the different symbols and a coding engine which encodes the symbols depending on their estimated probability. The probability distribution is considered to be known and only the actual arithmetic encoding process is contemplated in this paragraph.

4.2.1 Encoding to real Numbers

A theoretical examination of the arithmetic coding process is given with real numbers in this paragraph.

A data message is given as an input, consisting of symbols of an alphabet. For each symbol of the alphabet an estimation of its probability is given. The sum of all symbol-probabilities is 1.

The encoding process starts with an initial start interval, e. g. $I = [0\ 1)$. This interval will be divided into disjunct subintervals. Every symbol of the alphabet is mapped to a subinterval. The size of the subinterval matches the a priori probability of the symbol in relation to the start interval size. As an example: The alphabet consist of the symbols $S \in \{A, B, C, D\}$ with the a priori probabilities $Pr(S = s_i)$

$$Pr(S = A) = 0.5, Pr(S = B) = 0.25, Pr(S = C) = 0.125, Pr(S = D) = 0.125.$$

The division of the interval is illustrated in Fig 4.1 on the left. As you can see in the figure, the message ACB gets encoded.

Now the subinterval according to the actual symbol to encode is chosen as the new interval. In this case symbol A and interval [0 0.5). In the next step the interval is again divided into subintervals and the subinterval which belongs to the actual symbol

to be encoded becomes the new actual interval. Thus the interval gets smaller in every encoding step.

This iterative procedure will be done for every symbol of the input data message and can be described by the following equations [12]:

The start interval $I = [v_0, v_n]$ with the symbol index i, the alphabet size n and $v_0 = 0, v_i = \sum_{k=1}^{i} Pr(s_{k-1})$ is divided into the disjunct subintervals

$$I_i = [v_{i-1}, v_i).$$

The following variables are introduced for the encoding process: l is the control variable for the iteration process, L is the number of input symbols, and $u[l-1]$ is the lower bound of the current interval. $\Delta[l-1]$ describes the a priori probability of the input code word up to the position $l-1$ and in this case the actual size of the interval.

Initialization: $u[0] = 0, \Delta[0] = 1$

Actual symbol to encode: $S_l = s_{i_l}$, i_l correspond to the current input symbol index

Iteration:

$$\begin{aligned}
&l = 1(1)L : \\
&i = i_l \\
&u[l] = u[l-1] + \Delta[l-1] \cdot v_{i-1} \\
&\Delta[l] = \Delta[l-1] \cdot Pr(S_l = s_i) = \Delta[l-1] \cdot (v_i - v_{i-1}) \\
&\text{NEXT } l
\end{aligned}$$

At the end of this procedure the following interval is obtained ($u = u[L]$ and $\Delta = \Delta[L]$)

$$I = [u, u+\Delta),\ with\ u \in [0,\ 1), u+\Delta \in [0,\ 1)\ and\ \Delta = Pr(S_1 = s_{i_1}, ...S_L = s_{i_L}).$$

For highly probable symbols the interval gets only a little smaller, but for very unlikely symbols a very short interval is obtained. The smaller the interval the more bits are needed to describe a number within the interval. So a very unlikely sequence will need many bits and a highly probable sequence will need only few bits.

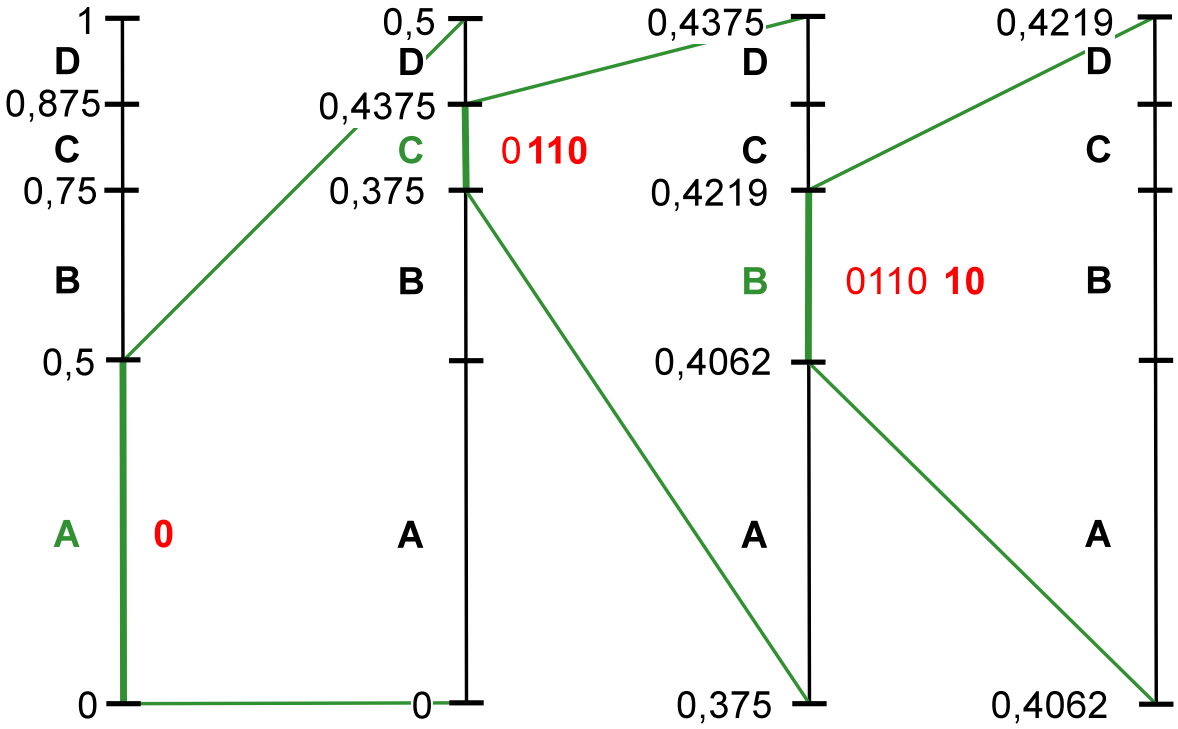


Figure 4.1: Arithmetic Encoding Process

For unique decoding, it is actually sufficient to represent the interval by a number within the interval

$$z \in I,\ u <= z < u + \Delta$$

and to know the number of encoded symbols.

The optimal code word length would be $n_{opt} = -ld(\Delta)$ after Shannon's source coding theorem [13]. z will be notated as a binary fraction number to get binary code symbols c_i.

$$z = 0.c_1c_2...c_n = \sum_{i=1}^{n} c_i 2^{-i}$$

The bits in red illustrate the codeword in Fig 4.1. The codeword consists of $n = \lceil -ld(\Delta) \rceil + 1 < n_{opt} + 2$ Bits $(n > n_{opt})$ [3, 12]. Thus it is nearly at the optimum and the difference to n_{opt} is negligible for a large n.

4.2.2 Encoding to Integer Numbers

Any kind of processors works with finite numbers. Arithmetic coding in theory needs infinite accuracy. Thus a few restrictions have to be introduced for real implementations. The M-ary Arithmetic Coder, 2^M denotes the alphabet size, in this work is based on an integer implementation [3] and it is integrated in the arithmetic coding framework which is introduced in the introduction paragraph on page 2.
Instead of symbol probabilities, the context model is based on frequencies.
The boundaries of the interval are 32-bit integer values. To ensure the accuracy if the boundaries get smaller, they have to be resized if the interval becomes half of the size of the start interval. As long as the the interval is less than half of the size of the starting interval, it has to be multiplied by the value of two and one bit is saved to the codestream.
Nevertheless there is some waste of bits by comparison to a theoretical approach with real infinite numbers.

4.2.3 Implementation Details

The input data is divided into unique blocks for a parallel processing similar to the binary arithmetic coding implementation at page 22. The frequencies are computed at the encoder and saved into a histogram. It is rather difficult to handle an own histogram for every block on the GPU because the demand of memory would be too big. Thus only one histogram for all blocks is computed. This makes it also possible to transmit the whole histogram to the decoder without having much overhead.
For storing the histogram as a side information, it is assumed that symbols with small absolute values are very frequent. This values around the zero valued symbol are saved as 32 bit unsigned integer values. Symbols with higher values which are less frequent are coded by a kth order Exp-Golomb code. The number of symbols is known at the encoder and at the decoder. The frequencies of the symbols are saved one after another with increasing distance to the zero valued symbol as integer values. If the

cumulative number of remaining frequencies is less than $2^{16} = 65536$, the remaining frequencies of the symbols are coded by the kth order Exp-Golomb code with $k = 1$ till the cumulative frequency is zero.

Other side information is coded and saved in the same way as described in the meta data section of the parallel binary arithmetic coder at page 31.

Chapter 5

CABAC

CABAC represents the state of the art entropy coder in terms of coding efficiency and speed. It is one of two entropy coding modes in the H.264/AVC standard and it is more complex with a better performance at the cost of speed in comparison to CAVLC which is the second mode.
The CABAC Framework is more than only an arithmetic coder. CABAC stands for **C**ontext-based **A**daptive **B**inary **A**rithmetic **C**oder. It consists of three main functional building blocks which are binarization, context modeling and binary arithmetic coding [11]. An Overview of this approach can be found in the block diagram of Fig 5.1.
A short guide of all parts is given in the following sections. A detailed explanation of the encoder can be found here [11]. The decoder is described in the H.264/MPEG-4 AVC standard [14].

5.1 Binarization

The Coding Engine processes only binary symbols, which are called bins. For that reason, M-ary symbols have to be binarized first. CABAC offers some binarization schemes, this explanation will be constrained to one scheme, called Unary/kth order Exp-Golomb (UEGk) binarization. This is a concatenation of the Truncated Unary

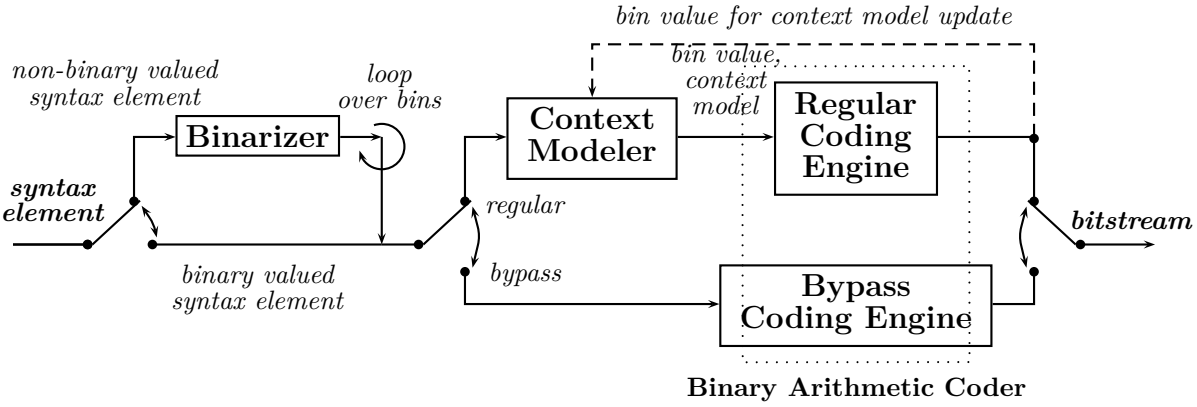


Figure 5.1: CABAC Block Diagram [11]

binarization and the kth order Exp-Golomb (EGk) binarization scheme.

Truncated Unary (TU) binarization builds the prefix part, where the input symbol is treated as a number with the value x. The TU binary representation of x is a variable length code with x times a "1" and a terminating "0". But if x is bigger then a predefined cutoff value S, only S preceding "1"s are used and a suffix part is needed to build a distinct code word.

The suffix part is a kth order Exp-Golomb binarization scheme which is also a concatenation of a prefix and suffix code word. The prefix part of the EGk code word consist of an unary code corresponding to the value of $l(x) = \lfloor log_2(x/2^k + 1) \rfloor$. The suffix code word of the EGk is the binary representation of $x + 2^k \cdot (1 - 2^{l(x)})$ using $k + l(x)$ significant bits.

Table 5.1 gives an example of this binarization scheme.

This binarization scheme is well suited for probability distributions, where symbols with values near 0 are highly probable and get unlikely for higher values. E. g. a highly peaked Laplacian distribution meets this requirement very well. Very few bins per symbol have to be encoded in average, if the distribution of the input sequence is close to that assumption. The less bins which have to be encoded, the faster the sequence can be encoded. Besides, the imprecision in computing is minimized because the symbol probabilities can be mapped to bin probabilities accurately if the estimated

x	Pr	Bin string									
		TU prefix									EG3 suffix
0	1/2	0									
1	1/4	1	0								
2	1/8	1	1	0							
3	1/16	1	1	1	0						
4	1/32	1	1	1	1	0					
...	...										
8	1/256	1	1	1	1	1	1	1	0		
9	...	1	1	1	1	1	1	1	1		0000
10	...	1	1	1	1	1	1	1	1		0001
11	...	1	1	1	1	1	1	1	1		0010
...	...										
bin index:		1	2	3	4	5	6	7	8	9	...
context index:		0	1	2	3	4	5	6	6	6	-

Table 5.1: UEG3 Binarization Scheme with cutoff value 9

distribution is near to an exponential distribution.

5.2 Context Modeling

Context modeling depends on the syntax element of course. A syntax element, in context of H.264/AVC, is a specific data type e.g. transform coefficients or predicted motion vectors.

CABAC has many predefined contexts which are all tabulated in the standard. A context defines a probability estimation of a bin, the probability if the bin is zero or one. The exact structure of a context is described in the next paragraph about binary arithmetic coding.

There are different approaches how a context is assigned to a specific bin, one is illustrated in table 5.1. As already explained in the last section, only the prefix part of the bin string is encoded by the regular coding engine. Thus only the bins of the prefix part have a corresponding context.

The first bin in this binarization scheme is the most important bin, cause it is very likely that this bin is zero and if that is the case, it is the only bin in the bin string.

The first bin has three different contexts and thus, the probability estimation has to be very accurate. The context is chosen depending on the input symbols of the near neighborhood. If the symbols in the neighborhood, which has already been processed, have very small absolute values, a context is chosen, where zero is highly probable. If the neighborhood has relatively big values, a context is chosen where the "1" has a higher probability. The neighboring symbols have to be correlated so that this approach make sense.
The bins with index $2-4$ have a unique context, thus no additional computation is needed. But there is also a second advantage by restricting the number of context per bin. The arithmetic coding engine adapts automatically the probabilities of the contexts. The more contexts, the slower the adaptation. A fast adaptation on the actual symbol probability distribution is an elementary condition for an efficient entropy coder. That also explains why the last five bins share one context. This bins occur very rarely and for the suffix bins, an adaptation would make no sense, that is why they are encoded by the fast bypass engine. This is a simplified coding engine where the symbol probabilities are assumed to be uniformly distributed.

5.3 Binary Arithmetic Coding

Binary arithmetic coding is used in almost every modern image and video coder. As an advantage over m-ary arithmetic coders, binary coders need only one probability estimation $Pr(s_0)$. The probability $Pr(s_1)$ equals $1 - Pr(s_0)$. Thus adapting the probabilities is very fast and easy. Besides there exist very fast implementations for binary arithmetic coders which do not need any multiplications or divisions.
The symbols "0" and "1" are not encoded directly, the context holds the information which symbol is the current most probable symbol (MPS). So it will be encoded a MPS or a LPS (least probable symbol).
In binary arithmetic coding the current interval is defined by a lower bound L and its width (range) R. The subintervals are two ranges, the range for the LPS (R_{LPS}) and

the range for the MPS (R_{MPS}). The ranges are computed as follows

$$R_{LPS} = R \cdot p_{LPS}$$

$$R_{MPS} = R - R_{LPS}.$$

The multiplication is the main bottleneck in terms of throughput for hardware implementations [11]. Binary Arithmetic Coders like the MQ coder (used in JPEG 2000) approximate this multiplication and set the range R to a fixed value [15].
The binary arithmetic coding scheme in CABAC is called modulo coder (M coder), which is a table-based approach.
The range R is approximated by a quantized value using an equi-partition of the whole range $2^8 <= R < 2^9$ into four cells. The index ρ of the current cell can be easily calculated by bit operations ($\rho = (R >> 6)\&3$) as it can be seen in Fig 5.2. The approximation is used only for the multiplication.
The probability of the LPS can attain only one of the 64 values p_σ which are computed as follows [11]

$$p_\sigma = \alpha \cdot p_{\sigma-1} \; \forall \; \{\sigma \in \mathbb{N} \mid 1 <= \sigma <= 63\}.$$

$$\text{with } \alpha = \left(\frac{0.01875}{0.5}\right)^{1/63} \text{ and } p_0 = 0.5.$$

The result of the multiplication with the range R_ρ and the p_σ is pre-computed and tabulated, which leads to the equation

$$R_{LPS} = R_\rho \cdot p_\sigma = TabRangeLPS[\sigma, \rho].$$

The loss of coding efficiency by this approximation is negligible in comparison to a conventional binary arithmetic coder [11].
The values of R and p_σ are illustrated in Fig 5.3. The whole binary decoding process is described in [14] in detail and the encoder is shown in Fig 5.2 and in [11].
p_σ is addressed only by its index σ as you can see in Fig 5.2 which is a value between 0 and 63 and can be represented by a 6-bit value. The information which binary symbol is

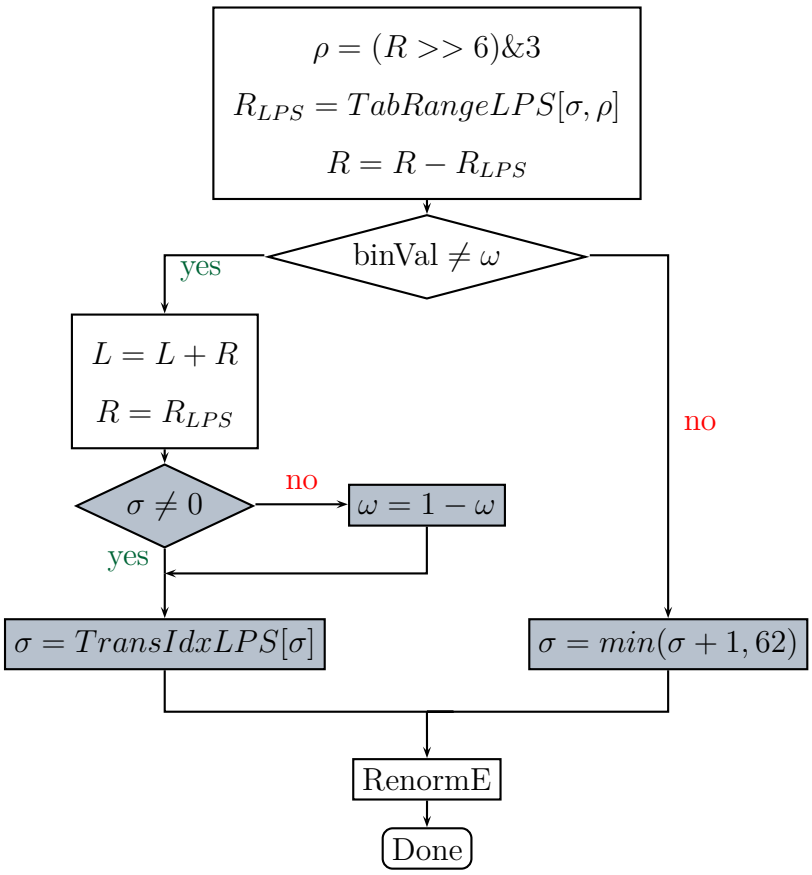
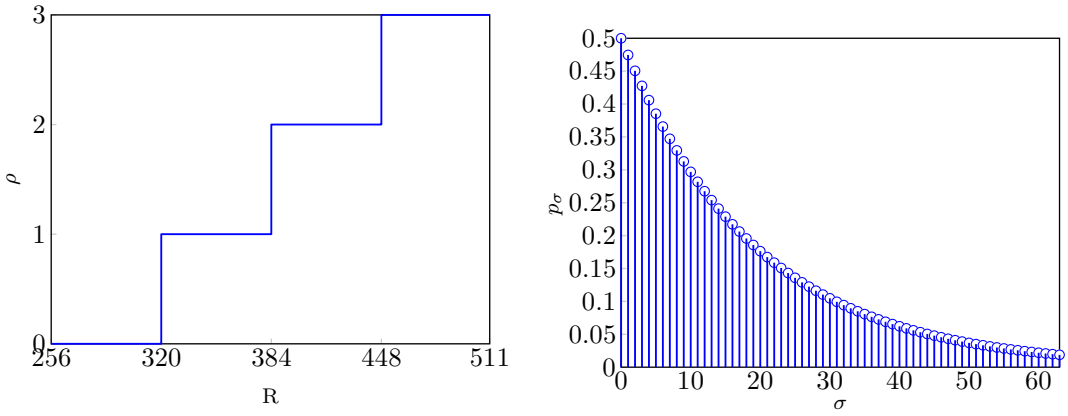


Figure 5.2: CABAC: Regular Coding Engine[11]

mapped to the MPS (ω) needs one bit. Thus the context described in context modeling section is a 7-bit value consisting of ω and σ.

Besides a unusual LPS takes more effort to code than a MPS which leads to a higher throughput in average. The grayly shaded boxes in Fig 5.2 are for the adaptation of the probabilities after every coded bin. By coding a MPS, σ is simply incremented by one up to a maximum value of 62 ($\sigma = 63$ is reserved for a special exit value). The adapted value of σ after a LPS is coded is stored in a table called TransIdxLPS. If $\sigma = 0$, the estimated probabilities of the LPS and MPS gets equal and the value of ω will be inverted.



(a) range (R)

(b) Probability LPS (p_{LPS})

Figure 5.3: CABAC: range R and p_{LPS} depending on their indices ρ and σ

Chapter 6

Parallel Binary Arithmetic Coding

6.1 General

Causal dependencies between the symbols to encode within the arithmetic coding process make a straight forward parallelization not possible.
There is some literature about to parallelize the arithmetic coding process. Many of the proposals restrict the parallelism to the encoder and also parts of the encoder have to be done sequentially like "Arithmetic coding in parallel" [16] or "Parallel design of arithmetic coding" [17]. This work is about the encoding and decoding processed in parallel.
Other approaches like MP-CABAC (Massively Parallel CABAC) [18] concentrate on hardware implementations and on a parallelism mainly at a very low level of arithmetic operations wich is not suitable for parallel processing on GPUs. Complete programming procedures have to be parallelized and the program flows have to be similar in CUDA to achieve a high throughput.
In this work the parallelization is done by coding unique blocks. So not the procedure is parallelized. But the input data is divided in blocks and every block is coded independently in parallel. This has some disadvantages over sequential arithmetic coding:

- Up to two bits are lost in comparison with the optimal code word length. If the

input data is divided in N blocks, up to $N \cdot 2$ bits are wasted.

- The size of the codestream can not be determined at the decoder and pointers have to be transmitted to indicate where the codestream of a certain block begins.

- Coding efficiency will be reduced by loosing the probability adaptation between neighboring blocks.

Techniques are presented in the following sections to minimize the supplementary data. The parallel binary arithmetic coder (PBAC) of this work is based on the CABAC framework. It also consist of binarization, context modeling and binary arithmetic coding. The differences to CABAC are clarified in own sections.
Two implementations have been created, one for the CPU and one for the GPU, but with an equal algorithm to make a reasonable comparison possible. Both were implemented for this work and are running by the framework which was already mentioned at the end of the introduction paragraph on page 2. Encoder and Decoder of the GPU implementation are based on the CudaImageToImageFilter filter from the CITK. The CPU implementations for encoder and decoder are derived from the GPU encoder and decoder and process the blocks sequentially.

6.2 Block Processing

How the blocks should be arranged that they could be coded in an efficient way depends on the type of data. This work is about CITK and CITK is about processing image data in two or three dimensions. Thus images are divided in two-dimensional blocks with a defined blocksize to capitalize on local dependencies of the probability distribution within the image. The input blocksize will be saved as a side information and is equal for every block to minimize the additional data. Blocks at the boundaries are simply cropped.
The blocksize should be as big as possible to minimize the size of supplementary data

in comparison to the size of the codestream. But, in contrast, to achieve a high parallelization which is necessary to get a high throughput, it is important to have a small blocksize. A balance have to be found for this two oppositional aims. In the ideal case, a blocksize can be found where the overhead is negligible and the capacity of the GPU is fully utilized. A big amount of data has to be processed in parallel to reach this case, because in this case, the capacity of the GPU is still fully utilized even if the blocksizes are very big and if the blocksizes are big, the size of the supplementary data is relatively small in relation to the size of the whole codestream.
A further way to preserve the locality is to arrange the input symbols in a way that the average two-dimensional distance of subsequent input symbols is as small as possible. Space Filling Curves do have this characteristic [19]. In this work a discrete approximation of the Hilbert curve [20] is implemented.

6.2.1 Hilbert Curve Scan Path

The arithmetic coding engine needs a one-dimensional data stream to code the symbols one by one. The discrete approximation of the hibert curve maps a two-dimensional grid of $n \cdot n$ discrete data points (block) to a linear data stream and preserves locality very well. This scan path is illustrated in Fig 6.1 on an prediction error image (a short explanation on predictive coding is given in Appendix A at page 50). Every data point (pixel) is represented by one gray square. The brightness of the square determines the pixel value. A white square determines the highest (positive) value a black square the smallest (negative) value.
A algorithm which performs this mapping is given in [20], a more detailed mathematical examination is given in [19].
The complexity to compute the scan points depends on the blocksize n. The Hilbert Curve is a recursive system where $log_2(n)$ recursion steps or iterations are needed to compute a single scan point. For large block sizes the complexity would be greater than the complexity of the arithmetic coding. Thus a fast implementation is needed.

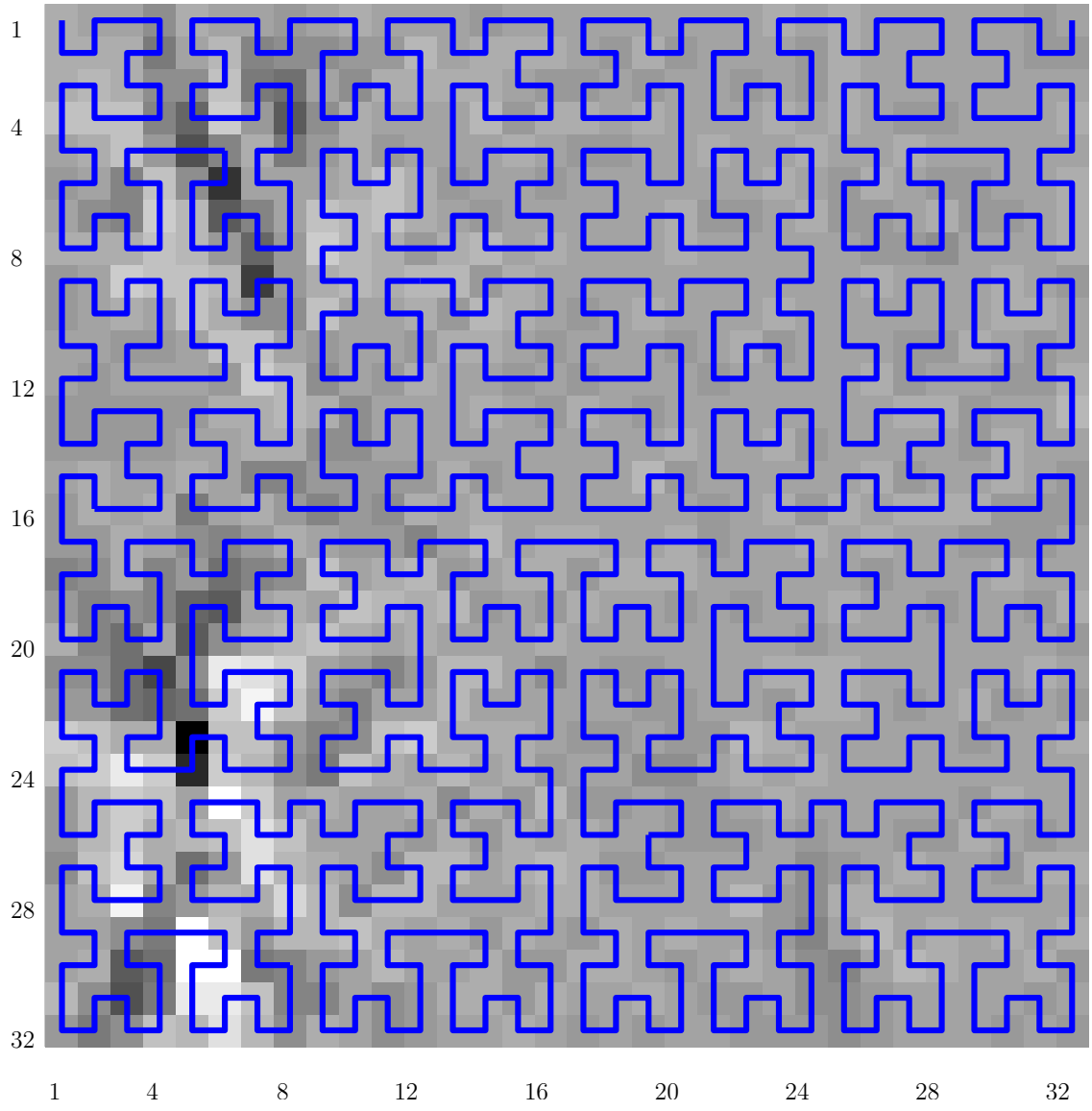


Figure 6.1: Hilbert Curve Scan Path of an prediction error image block with $32 \cdot 32$ pixel. The brighter the gray levels, the bigger the values of the gray levels. This are signed values, the negative minimum is black and the positive maximum is white.

To simplify the computing of every single data point, small blocks (e. g. $16 \cdot 16$ data points) are precomputed. Only four different scan orders are possible. As a consequence of the recursive system, the whole block can be represented by shifted versions of the four small blocks.

In 6.1 you can see three of the four possible $16 \cdot 16$ subblocks. The first starts at (1, 1) and ends at (1, 16), so its starts at the left side and ends at the left side. For that reason it has the orientation *west*. The second block starts at (1, 17) and ends at (16, 17), so it has the orientation *north* and is shifted by (0, 16). The third block also have the orientation *north* and the last block starts at (32, 16) and ends at (32, 1) which leads to the orientation *east*. The change of the orientation is called rotation.

The subblocks are in the same order as the scanning order within the subblock (recursive system). This means, the scanning path of the consecutive subblocks is the same as the scanning path of the data points within a subblock, with one difference: Within a subblock the distance between to consecutive scanned data points is one and the distance between two consecutive scanned subblocks is the subblock size.

The rotation of the orientation is also a recursive system with the following four different basic cycles (0 = west, 1 = north, 2 = east, 3 = south):

0112

1003

2330

3221

If only up to four subblocks are present, the first basic orientation rotation cycle 0112 describes the consecutive orientations.

If up to 16 subblocks are present, starting with orientation west, the consecutive orientations would be:

0112 **1**003 **1**003 **2**330

In general this is described by the pseudo-code of GetOrientationOfSubblock.

Blocks of higher order can be computed very fast with the four subblocks and the four basic orientation rotation cycles.

Procedure GetOrientationOfSubblock(x)

Data: Index of the current subblock (x), array of basic orientation cycles ($orientArray$)

Result: Orientation of the current subblock ($orient$)

$n = 0$;

if $x > 0$ **then**

  $n = \lfloor log_4(x) \rfloor$;

end

$idx0 = 0$;

$orient = 0$;

while $n >= 0$ **do**

  $idx1 = \lfloor x/4^n \rfloor$;

  $orient = orientArray[idx0, idx1]$;

  $idx0 = orient$;

  $x = x - 4^n \cdot idx1$;

  $n = n - 1$;

end

Implementation CPU

The scanning positions of the whole block are computed once for every block and are saved in two one-dimensional arrays for the x- and y-direction.

Implementation GPU

The scanning positions of the four subblocks are computed once for every CUDA thread block and are saved in eight one-dimensional arrays on shared memory. The scanning positions are computed in parallel. Thus every thread in a thread block have to compute only a few scanning positions. If the subblock size is 16, $4 \cdot 16^2 = 1024$ scanning positions have to be computed in total and if the thread block size is 128, $1024/128 = 8$ scanning positions per thread have to be computed.
Typical blocksizes are 64 or 128, a block size of 64 would lead to 4096 scanning positions. So every thread would have to compute 32 scanning positions (by a CUDA thread block size of 128). The effort would be acceptable but the need of shared memory would be to high. Four times more shared memory would lead to four times less resistent warps per multiprocessor (if shared memory is the limiting factor).

6.3 Binary Arithmetic Coding

The binary arithmetic coding process is almost the same like in the CABAC framework. The bypass coding engine is identical to the one in CABAC. The regular coding engine is not using tabulated values because arithmetic operations are much faster than memory accesses on the GPU. The binary symbols $S = \{0, 1\}$ are coded directly and they are not mapped to LPS and MPS. This means

$$R_{S=1} = R \cdot Pr(S = 1)$$

is computed arithmetically, no approximation is used.

The probability adaptation is done as follws:

$$Pr(S=1)_{new} = \begin{cases} max(\alpha \cdot Pr(S=1)_{old},\ 0.005), & \text{if } S=0 \\ min(\alpha \cdot Pr(S=1)_{old} + (1-\alpha),\ 0.995), & \text{if } S=1 \end{cases}$$

Because if $S=1$ and $Pr(S=1) < 0.995$

$$Pr(S=0)_{new} = \alpha \cdot Pr(S=0)_{old} = \alpha \cdot (1 - Pr(S=1)_{old})$$

$$Pr(S=1)_{new} = 1 - Pr(S=0)_{new} = 1 - \alpha \cdot (1 - Pr(S=1)_{old})$$

$$Pr(S=1)_{new} = \alpha \cdot Pr(S=1)_{old} + (1-\alpha).$$

The minimum and maximum values are needed to avoid to exceed the precision of 32 bit floating point values.

α do not have to be a fixed value like in the table-based CABAC approach. A proper balance has to be found between an acurrate estimation and a fast adaptation depending on the input sequence.

6.4 Binarization and Context Modeling

The initial probability estimation of a block has to be very accurate to compensate, as good as possible, the advantage of a precise estimation which can be achieved by a continuous adaption on the whole input sequence coded sequentially.

Proper context modeling depends on the characteristics of the data. Different approaches are needed for different types of data. But the probability distribution of the input data has to be highly peaked to achieve high coding gains.

The laplace distribution with a mean value of zero and a computed standard deviation is a good approximation for the probability distribution of many types of data with a highly peaked distribution. One of them are prediction error images like in Fig 6.1 which are used in the paragraph about the results for testing the algorithm (a short

explanation on predictive coding is given in Appendix A at page 50).
The initial context modeling is done by estimating the symbol probabilities by a discrete version of the laplace distribution. This distribution is axially symmetric. It can be assumed that symbols with the same absolute values have almost the same probability of occurrence. Thus it is reasonable to code the sign value as an independent bit to attain a faster adaptation by this alphabet reduction.
The binarized absolute values are coded by the regular coding engine and the sign bit is coded by the bypass engine. The Unary/kth order Exp-Golomb (UEGk) binarization scheme, explained in the paragraph about CABAC, is used to binarize the absolute values.
The number of contexts n depends on the initially estimated symbol probabilities. n is equal to the maximum number of symbols with the smallest absolute value which have a cumulative probability less than p_{cumMax}. p_{cumMax} is a predefined value like 0.9.
The first $n-1$ bins do have a distinct context. The remaining unlikely bins of the unary prefix share the last context. The cutoff value is predefined. The context in this approach is a 32-bit floating point value describing the estimated probability of the bin and is adapted as explained in the last section.
The bins of the kth order Exp-Golomb suffix are coded by the bypass engine.
So only the most probable symbols are taken into account for the context modeling, similar to the CABAC framework.
The first step of the block processing is the computation of the standard deviation for the input symbol values with a mean value of zero. This is done only at the encoder and the standard deviation is transmitted to the decoder. Then the number of contexts can be computed. The last step of the initial context modeling of a block is the estimation of the probabilities of the bins which is given by the procedure computeUEGkContexts. The maximum number of contexts is restricted to eight because on the GPU the contexts reside at the very limited shared memory.

Procedure computeUEGkContexts(n, symbolProbArray, contextArray)

Data: number of contexts (n), array of input symbol probabilities (symbolProbArray)

Result: array of contexts (contextArray)

cumProb = 0;

for $i = 0$ **to** $n - 1$ **do**

(The estimated symbol probabilities are computed on the fly normally. They are accessed through an array only for illustration);

contextArray[i] = 1 − symbolProbArray[i]/(1 − cumProb);

contextArray[i] = min(contextArray[i], 0.005);

contextArray[i] = max(contextArray[i], 0.995);

cumProb = cumProb + symbolProbArray[i];

end

6.5 Metadata

Diverse side information has to be transmitted within the codestream as already explained. The goal is to use as few space as possible. The side information is stored at the beginning of the coded data and consists of the following parts: coding mode, blocksize, images sizes in three dimensions, standard deviation for every block and the sizes of the codestream per block.

6.5.1 Coding Mode

The first byte determines the coding mode. Different modes are defined by an enumeration data type, e. g. bac_TUEGk_HilbScan which is the binary arithmetic coder as described in this paragraph with a UEGk binarization and with a Hilbert curve scanning path. The last four bytes can be used to determine a prediction mode, if the last four bytes are zero, no prediction is used (a short explanation on predictive coding is given in Appendix A at page 50).

6.5.2 Blocksize

Only two-dimensional square blocks are available because only two bytes for one block-size value for width and height is used. Typical blocksizes are $32 \cdot 32$, $64 \cdot 64$, or $128 \cdot 128$. If the block-size should cover the whole image, it has to be at least as big as the width and the height.

6.5.3 Image Sizes

Images with up to three dimensions can be coded. If the image is only two-dimensional, the size of the third dimension is set to one. Two bytes are used for each dimension, which restricts the maximal image size to $2^{16} = 65536$.

6.5.4 Standard Deviation per Block

Computations depending on the standard deviation are done by 32-bit floating point values. But the accuracy of the last bits is negligible. Two bytes are enough and for that reason the values are converted to 16-bit floating point values and half of the space is saved.

6.5.5 Codestream Size per Block

The codestream sizes per block are predicted by a Laplace distribution with the given standard deviation (a short explanation on predictive coding is given in Appendix A at page 50).

The first assumption is that the size of the codestream is about the size of the optimal codeword size depending on the entropy. The second assumption is that the Laplace distribution with the given standard deviation is a good approximation for the probability distribution of the input data.

Thus the codeword size can be approximated by an estimated entropy multiplied by the number of input symbols. The summation of the entropy formula is approximated

by an integral and instead of the discrete, the continuous Laplace distribution is used as a probability estimation.

The symbol alphabet $S \in \{s_i \in \mathbb{N} | -a \leq s_i < a\}$ with the symbol index $0 < i \leq 2a$ consists of signed (8-, 16-bit) integer values which leads to the following approximation for the entropy $H(s)$:

$$H(S) = -\sum_{i=1}^{2a} Pr(s_i) \cdot \log_2 \left(Pr(s_i)\right) \approx -\int_{-a}^{a-1} \frac{1}{2\sigma} e^{-\frac{|x|}{\sigma}} \cdot \log_2 \left(\frac{1}{2\sigma} e^{-\frac{|x|}{\sigma}}\right) dx =$$

$$-\frac{1}{2\sigma \cdot \ln(2)} \int_{-a}^{a-1} e^{-\frac{|x|}{\sigma}} \cdot \left(\ln\left(\frac{1}{2\sigma}\right) - \frac{|x|}{\sigma}\right) dx \approx$$

$$-\frac{1}{\sigma \cdot \ln(2)} \int_{0}^{a} e^{-\frac{x}{\sigma}} \cdot \left(\ln\left(\frac{1}{2\sigma}\right) - \frac{x}{\sigma}\right) dx$$

$$u = \ln\left(\frac{1}{2\sigma}\right) - \frac{x}{\sigma} \Rightarrow u' = -\frac{1}{\sigma}$$

$$v' = e^{-\frac{x}{\sigma}} \Rightarrow v = -\sigma \cdot e^{-\frac{x}{\sigma}}$$

$$-\frac{1}{\sigma \cdot \ln(2)} \left([u \cdot v]_0^a - \int_0^a -\frac{1}{\sigma} \cdot \left(-\sigma \cdot e^{-\frac{x}{\sigma}}\right) dx\right) =$$

$$-\frac{1}{\sigma \cdot \ln(2)} \left([u \cdot v]_0^a - \int_0^a e^{-\frac{x}{\sigma}} dx\right) =$$

$$-\frac{1}{\sigma \cdot \ln(2)} \left([u \cdot v]_0^a + \sigma \cdot e^{-\frac{a}{\sigma}} - \sigma\right) =$$

$$-\frac{1}{\sigma \cdot \ln(2)} \left(\left[-\left(\ln\left(\frac{1}{2\sigma}\right) - \frac{x}{\sigma}\right) \cdot \sigma \cdot e^{-\frac{x}{\sigma}}\right]_0^a + \sigma \cdot e^{-\frac{a}{\sigma}} - \sigma\right) =$$

$$-\frac{1}{\sigma \cdot \ln(2)} \left(-\left(\ln\left(\frac{1}{2\sigma}\right) - \frac{a}{\sigma}\right) \cdot \sigma \cdot e^{-\frac{a}{\sigma}} + \ln\left(\frac{1}{2\sigma}\right) \cdot \sigma + \sigma \cdot e^{-\frac{a}{\sigma}} - \sigma\right)$$

Assumption:

$$a >> \sigma$$

$$\Rightarrow \sigma \cdot e^{-\frac{a}{\sigma}} \approx 0$$

$$\Rightarrow -\left(\ln\left(\frac{1}{2\sigma}\right) - \frac{a}{\sigma}\right) \cdot \sigma \cdot e^{-\frac{a}{\sigma}} \approx 0$$

$$\Rightarrow H(S) \approx -\int_{-a}^{a-1} \frac{1}{2\sigma} e^{-\frac{|x|}{\sigma}} \cdot \log_2\left(\frac{1}{2\sigma} e^{-\frac{|x|}{\sigma}}\right) dx \approx \tilde{H}(S) = \frac{1}{ln(2)}\left(ln(2\sigma) + 1\right)$$

With the number of input symbols m, the codeword size n could be approximated by

$$n \approx \tilde{n} = \lfloor m \cdot \tilde{H}(S) \rfloor.$$

And only the residuum r of the prediction

$$r = n - \tilde{n}$$

is transmitted with one sign bit if $r \neq 0$ and the absolute value as a kth order Exp-Golomb codeword.
n can be recovered at the decoder by computing $\tilde{n}$ in the same way as at the encoder and by transposing the equation to

$$n = r + \tilde{n}$$

So, if the approximation is good and $|r|$ is very small, only a few bits have to transmitted because the smaller the $|r|$ the smaller the codeword.

Chapter 7

Experimental Results

Experiments are performed to evaluate the coding efficiency and the speed of the implementations. The main issue is to figure out what impact the parallelization does have on the coding efficiency and how big the speed-up is by running on a GPU. Prediction error images are used for test data (a short explanation on predictive coding is given in Appendix A at page 50). The source images are mainly 3-D volume images from medical modalities like CT or MRI scanners. Yet, also series of 2-D synthetic images are tested.

The test environment consists of a AMD Phenom(tm) II X6 1090T processor [21] with a clock frequency of 3.2 GHz and a Nvidia GeForce GTX 480 graphics card [6].

The first set of data are 8-bit prediction error volume images from medical modalities. The prediction mode is called edge-directed prediction [22] and was done with the numerical computing environment Matlab [23]. This is a prediction mode with good decorrelation characteristics. The images were chosen because of their different size and their different entropy.

All results concerning the entropy are computed with the entropy formula ($H(S) = -\sum_{i=1}^{n} Pr(s_i) \cdot \log_2 (Pr(s_i))$) with statistically dependent symbols s_i.

7.1 Impact of Blocksize

7.1.1 Explanations to the Results Tables

Table 7.1 shows some Information about the coding efficiency of different coding modes depending on the blocksize.
The coding mode PBAC stands for parallel binary arithmetic coding and in this case for the standard parallel, binary coder which was implemented and described in the previous paragraph. Every 2-D slice of the 3-D volume is divided into 2-D square blocks with the given blocksize and the voxels are scanned in the order of a discrete Hilbert path. The blocks are processed on the GPU in parallel by the binary arithmetic coding engine.
The algorithm of BAC is the same as PBAC but all processing is done on the CPU.
The coding mode CABAC indicates a standard CPU CABAC implementation derived from libcabac [24] and ported to the ITK. Block division and scanning is the same as in BAC mode.
The names of the volume images are in the first column of table 7.1 with a declaration of the width, height and depth of the image in brackets.
The column bits/symbol determines how much coded bits in average are needed for one 8-bit input symbol. The values are smaller than the entropy (also in bits/symbol) in every row. This is because the prediction error images still do have some local dependencies and the neighboring voxels are not statistically independent and with an adaptive coding mode which can adapt on the current probability distribution, it is possible to get smaller bits/symbol values (rate) than the entropy value. The saving column is computed by the following equation:

$$(\text{entropy} - \text{bits/symbol})/\text{entropy} \cdot 100\%.$$

The relation between overhead due to meta data and blocksize is shown at table 7.2. Size total indicates the size of the whole coded image including the size of the meta data which is specified in the next column. The overhead, in this case is the ratio of

meta data to the total size.
The last two tables, 7.3 for encoding and 7.4 for decoding, indicate the duration of the ITK filters and of the CUDA kernels which are coding the volume images by different modes and blocksizes.

7.1.2 Interpretations of the Results Tables

As assumed in the paragraph about parallel binary arithmetic coding at page 22, more bits are needed for smaller blocksizes. But it was the goal to minimize this difference. 9 bytes are needed for storing the coding mode, the blocksize and the image sizes as meta data. In a naive implementation 4 bytes for the standard deviation and 4 bytes for the blocksize, 8 bytes in total, would be needed for every block. The A0 volume image with a blocksize of 64 consists of 8000 blocks. This means $9\text{B} + 8000 \cdot 8\text{B} = 64009\text{B}$ would be needed instead of 24345B (table 7.2) which is 2.6 times as much in comparison to this implementation. The savings with other images and blocksizes are very similar.
Blocksizes of 64 or 128 lead to very good results as it can be seen in table 7.1 and are as good as or better than the CABAC implementation concerning the coding efficiency. The coding efficiency of bigger blocksizes increase very moderate but the coding speed decreases heavily which makes blocksizes of 64 or 128 a very good choice. Smaller blocksizes are not very suitable because the big overhead of meta data which is computed by the cpu leads to a relatively high total coding time. The biggest increase of speed of the GPU implementation in comparison to the two CPU implementation can be seen at the very big A3 image. The parallel GPU implementation is more than 39 times faster than in the sequential BAC and CABAC mode and needs for the 1210 slices only 1.1 second for the whole ITK filter. But for the small N31 image with only 16 slices such a big speed up is not possible.
The GPU implementation needs a high amount of data to be processed in parallel to achieve a high throughput and good coding efficiency.
This CABAC-like coding approaches achieve noticeable better coding efficiencies than

computed by the entropy formula even though the edge-directed prediction is very efficient in decorrelating image data.
Their are two reasons why the BAC provides better results than the CABAC. Firstly, no approximations are used for computations in the approach of this work in comparison to CABAC. Secondly, the adaptation in BAC is more accurate but slower than in CABAC. This is because of the adaptation parameter α which is in CABAC a fixed value of approximately 0.94 and in (P)BAC 0.98. Many different syntax elements with many contexts have to coded by the CABAC engine without an acurrate initial probability estimation. Thus is it important to adapt very fast. In this approach, only prediction error images are coded with only a few contexts and a very accurate initial probability estimation is available. Thus it is better to have a more accurate adaptation than in the CABAC approach. This approach do have also the flexibility to change the parameter α for every syntax element which is not possible by a table-based arithmetic coder.

7.2 Image series coding

The arithmetic coding framework provides the possibility to code whole image series. A prediction filter with an prediction error image as output can be executed before running the arithmetic coding filter and extends the framework to a lossless image series coder.
Table 7.5 shows some results of this image series coder. The sequences are predicted by a gradient adjusted predictor (GAP) [25]. The prediction is computed on the GPU with the same block partition like the arithmetic coder. The entropy column denotes the entropy of the prediction error image sequence.
This test demonstrates that it is possible to code videos like the Big Buck Bunny, losslessly, approximately in real-time with this arithmetic coding framework. The input video has the size of 1920x1080 pixel and consists of three channels without any sub-sampling.

Volume image	Mode	Blocksize	Entropy	Bits/Symbol	Savings
N31 (512x512x16)	PBAC	32	0.9278	0.8540	7.96%
		64		0.8251	11.07%
		128		0.8182	11.82%
		256		0.8178	11.86%
	BAC	512		0.8144	12.22%
	CABAC	512		0.8178	11.86%
A0 (512x512x125)	PBAC	32	1.6908	1.6205	3.80%
		64		1.5930	5.78%
		128		1.5859	6.20%
		256		1.5843	6.30%
	BAC	512		1.5838	6.33%
	CABAC	512		1.5937	5.74%
N0 (512x512x448)	PBAC	32	1.3746	1.3000	5.43%
		64		1.2723	7.44%
		128		1.2653	7.95%
		256		1.2630	8.12%
	BAC	512		1.2622	8.18%
	CABAC	512		1.2686	7.71%
A3 (512x512x1210)	PBAC	32	2.5322	2.4359	3.95%
		64		2.4078	4.91%
		128		2.4007	5.19%
		256		2.3983	5.29%
	BAC	512		2.3978	5.31%
	CABAC	512		2.4165	4.57%

Table 7.1: Entropy of prediction error volume images of medical modalities and Bits/Symbol after coding with different modes depending on blocksize. Savings of Bits/Symbol in relation to the entropy

Volume image	Blocksize	Size Total	Size Meta Data	Overhead
N31 (512x512x16)	32	531 680B	14 950B	2.81%
	64	513 704B	3 946B	0.77%
	128	509 383B	1 081B	0.21%
	256	509 140B	284B	0.06%
	512	507 069B	80B	0.02%
A0 (512x512x125)	32	6 637 604B	96 188B	1.45%
	64	6 524 849B	24 345B	0.37%
	128	6 495 927B	6 556B	0.10%
	256	6 489 371B	1 798B	0.03%
	512	6 487 313B	531B	0.01%
N0 (512x512x448)	32	19 083 667B	344 310B	1.80%
	64	18 677 563B	88 950B	0.48%
	128	18 574 067B	24 132B	0.13%
	256	18 540 323B	6 533B	0.04%
	512	18 528 868B	1 843B	0.01%
A3 (512x512x1210)	32	96 580 867B	930 701B	0.96%
	64	95 466 289B	235 132B	0.25%
	128	95 184 081B	63 305B	0.07%
	256	95 092 367B	17 503B	0.02%
	512	95 069 442B	4 830B	0.01%

Table 7.2: Overhead due to meta data (coding mode, blocksize, image size, per block: size of coded block, standard deviation) of prediction error volume images of medical modalities depending on blocksize

Volume image	Mode	Blocksize	D. Kernel	D. Filter
N31 (512x512x16)	PBAC	32	0.01s	0.10s
		64	0.03s	0.11s
		128	0.13s	0.20s
		256	0.48s	0.55s
	BAC	512	n.a.	0.44s
	CABAC	512	n.a.	0.43s
A0 (512x512x125)	PBAC	32	0.06s	0.22s
		64	0.07s	0.18s
		128	0.17s	0.26s
		256	0.61s	0.70s
	BAC	512	n.a.	3.48s
	CABAC	512	n.a.	3.35s
N0 (512x512x448)	PBAC	32	0.19s	0.59s
		64	0.21s	0.41s
		128	0.24s	0.40s
		256	0.60s	0.74s
	BAC	512	n.a.	10.69s
	CABAC	512	n.a.	10.39s
A3 (512x512x1210)	PBAC	32	0.58s	1.56s
		64	0.64s	1.10s
		128	0.76s	1.09s
		256	0.99s	1.28s
	BAC	512	n.a.	43.24s
	CABAC	512	n.a.	42.31s

Table 7.3: Duration of encoding prediction error volume images of medical modalities with different modes depending on blocksize

Volume image	Mode	Blocksize	D. Kernel	D. Filter
N31 (512x512x16)	PBAC	32	0.01s	0.08s
		64	0.02s	0.09s
		128	0.07s	0.14s
		256	0.26s	0.34s
	BAC	512	n.a.	0.33s
	CABAC	512	n.a.	0.32s
A0 (512x512x125)	PBAC	32	0.06s	0.14s
		64	0.05s	0.13s
		128	0.10s	0.18s
		256	0.38s	0.46s
	BAC	512	n.a.	2.43s
	CABAC	512	n.a.	2.36s
N0 (512x512x448)	PBAC	32	0.02s	0.26s
		64	0.16s	0.25s
		128	0.16s	0.25s
		256	0.36s	0.45s
	BAC	512	n.a.	7.47s
	CABAC	512	n.a.	7.22s
A3 (512x512x1210)	PBAC	32	0.68s	0.85s
		64	0.70s	0.84s
		128	0.77s	0.90s
		256	0.62s	0.75s
	BAC	512	n.a.	31.95s
	CABAC	512	n.a.	30.36s

Table 7.4: Duration of decoding prediction error volume images of medical modalities with different modes depending on blocksize

Frames	Entropy	Bits/Symbol	Savings	Encoding	Decoding
50 - 79	1.9882	1.6348	17.77%	1.04s	0.78s
554 - 583	4.6362	4.3680	5.78%	1.48s	1.02s

Table 7.5: Results of coding sequences of the Big Buck Bunny video [26] (Blocksize 64, parallel binary arithmetic coding, gradient adjusted predictior [25])

The first sequence (frames 50 -79) has not much details in relation to the second sequence which is of the part with the most details in the whole video. Thus, the first sequence can be compressed much better than the second video. The execution time of the second sequence is also worse. This is because the number of bits which have to be processed by the binary arithmetic coding engine is much higher in the complex second sequence. The encoding kernel of the first sequence takes 0.42 seconds and 0.80 seconds for the second sequence. This shows the dependence of the processing time on the entropy of the input image. Lossy video codecs do have a much higher compression and thereby, binary arithmetic coding can be performed much faster.

Table 7.6 shows a short comparison with the orignial images from [26] coded by the Portable Network Graphics (PNG) format and by the JPEG 2000 coder OpenJPEG in lossless mode. The coding with GAP and PBAC achieves a better compression than the original PNG coded files for both sequences and a better compression the the JPEG 2000 coder for the first sequence. The compression with GAP and PBAC achieves a worse compression than the JPEG 2000 coder for the second sequence. The high entropy of the second sequence is one reason for the worse compression of the second sequence. The UEGk binarization scheme is not well suited for images with such a high entropy. The implementation of a another binarization scheme, used depending on the standard deviation of a block, could possibly improve the compression performance.

Mode	Frames	Size Codestream	Savings
PNG (original)	50 - 79	41 600 015B	0.00%
	554 - 583	108 300 072B	0.00%
JPEG2000	50 - 79	41 545 632B	0.13%
	554 - 583	97 426 823B	10.04%
PBAC + GAP	50 - 79	38 137 663B	8.32%
	554 - 583	101 898 075B	5.91%

Table 7.6: Comparison with the original Big Buck Bunny PNG images [26] and JPEG 2000 lossless compressed images (by OpenJPEG [27]). Savings are in relation to the original PNG files.

Volume image	Hilbert (Bits/Symbol)	None (Bits/Symbol)	Savings
N31 (512x512x16)	0.8182	0.8482	3.55%
A0 (512x512x125)	1.5859	1.6103	1.51%
N0 (512x512x448)	1.2653	1.2984	2.55%
A3 (512x512x1210)	2.4007	2.4296	1.19%

Table 7.7: Savings by using a Hilbert curve as scanning mode instead of scanning line by line (Blocksize 128, parallel binary arithmetic coding)

7.3 Impact of Hilbert Curve Scan

The scanning of the pixels / voxels in the order of a Hilbert curve is done to preserve localities. Table 7.7 shows that some savings can be obtained if such an adaptive procedure, like the parallel binary arithmetic coder implemented in here, is used.

Volume image	Mode	Blocksize	Entropy	Bits/Symbol	Savings
A0 (512x512x125)	PMAC	64	1.6908	1.7167	-1.53%
		128		1.7143	-1.39%
	MAC	512		1.7133	-1.33%
N0 (512x512x448)	PMAC	64	1.3746	1.4711	-7.02%
		128		1.4687	-6.85%
	MAC	512		1.4678	-6.78%

Table 7.8: Entropy of prediction error volume images of medical modalities and Bits/Symbol after coding with different modes depending on blocksize

7.4 Standard M-ary Arithmetic Coding

The test results of the M-ary arithmetic coder implemented in here are listed in the tables 7.8, 7.9, 7.10 and 7.11. The tables are composed in the same way as for the binary arithmetic coders. This M-ary arithmetic coder uses a precomputed histogram to compress the input data.

The coding modes are called M-ary arithmetic coding (MAC) and parallel M-ary arithmetic coding (PMAC).

The histogram is computed by the encoder on the cpu. Thus the encoder is relatively slow as it can be seen in table 7.10. Further speed up would be attainable if the histogram would be computed on the gpu.

This histogram stays the same over the whole coding process, no adaptation is performed. Thus it is not possible to get better results as the values computed by the entropy formula and here they are a bit worse but still close to the entropy.

The very little overhead due to the meta data is one reason for this result (see table 7.9 for details). Only one histogram is transmitted in a efficient way as explained at page 13 and the blocksizes are coded in the same way as in the binary mode. The decoding speed, listed in table 7.11 is comparable to the speed of the binary arithmetic coder.

Volume image	Blocksize	Size Total	Size Meta Data	Overhead
A0 (512x512x125)	64	7 031 510B	10 548B	0.15%
	128	7 021 743B	3 521B	0.05%
	512	7 017 765B	431B	0.01%
N0 (512x512x448)	64	21 595 317B	36 643B	0.12%
	128	21 560 671B	11 862B	0.06%
	512	21 547 196B	1 440B	0.01%

Table 7.9: Overhead due to meta data of prediction error volume images of medical modalities depending on blocksize

Volume image	Mode	Blocksize	D. Kernel	D. Filter
A0 (512x512x125)	PMAC	64	0.04s	0.49s
		128	0.07s	0.50s
	MAC	512	n.a.	1.97s
N0 (512x512x448)	PMAC	64	0.12s	1.54s
		128	0.10s	1.48s
	MAC	512	n.a.	6.52s

Table 7.10: Duration of encoding prediction error volume images of medical modalities with different modes depending on blocksize

Volume image	**Mode**	**Blocksize**	**D. Kernel**	**D. Filter**
A0 (512x512x125)	PMAC	64	0.03s	0.10s
		128	0.08s	0.16s
	MAC	512	n.a.	2.36s
N0 (512x512x448)	PMAC	64	0.16s	0.25s
		128	0.11s	0.19s
	MAC	512	n.a.	10.44s

Table 7.11: Duration of decoding prediction error volume images of medical modalities with different modes depending on blocksize

Chapter 8

Conclusion

The parallelization of the arithmetic coding is based on block partition. The implemented framework makes it possible to encode and decode 2-D or 3-D images and videos very fast on a GPU by using the CITK.

Only minimal loss in coding efficiency occurs compared to a sequential execution if the blocksize is not too small. Smart coding of the meta data leads to very little overhead. A fast implementation of a Hilbert curve path scan was developed and by adapting the symbol probabilities to the local probability distribution in a fast and accurate way, noticeable savings can be achieved in comparison to the bits/symbol computed by the entropy formula.

The Unary/kth order Exp-Golomb (UEGk) binarization used in this work is very well suited for high peaked distributions. Further improvements would be possible by implementing other binarization schemes for flatter distributions. The binarization scheme could be chosen depending on the standard deviation.

Sequential execution on the CPU was also incorporated to the framework. Three different modes are available, a M-ary integer implementation, a ported CABAC implementation and a binary floating point implementation which is very similar to the CABAC implementation. Instead of table look-ups, the binary arithmetic coder performs all arithmetic operations within the CUDA kernel. This make it possible to capitalize from the architecture of a GPU which is fast in performing floating-point

operations. Furthermore, memory usage is reduced and many memory accesses are avoided which both are not suitable for a GPU because of the small size of the fast onboard-memory.

A very high throughput can be achieved with a good coding efficiency by using the tremendous computational power of modern GPUs.

Appendix A

Predictive Coding

The implemented coding framework is configured for data with a highly peaked probability distribution. Images without any preprocessing normally do not have this characteristic but the neighboring pixels are highly correlated. The goal is to achieve a highly peaked distribution of the data values by decorrelating the data values. Predictive coding can be used for that issue. Pixel values are predicted from previously decoded neighboring pixels by exploiting the statistical dependencies [15]. If the predicted values are around the actual values, the differences between this two values are very small. This differences are called prediction error and if the prediction error values are mainly small values, the unequal distribution is achieved.

Only already coded data values are taken into account for the prediction. Thus it is possible to do the same predictive coding at the encoder and at the decoder. The actual value can be reconstructed by adding the prediction error value to the predicted value and because of that it is sufficient to transmit only the prediction error values. This values are well suited for coding with an entropy coder because the entropy is low and high coding gains can be achieved. But some local dependencies still remain which makes adaptive entropy coding very attractive.

List of Figures

List of Tables

Bibliography

[1] Wikipedia, "Moore's law — wikipedia, the free encyclopedia," 2012, [Online; accessed 18-September-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Moore%27s_law&oldid=512597275

[2] ——, "Parallel computing — wikipedia, the free encyclopedia," 2012, [Online; accessed 18-September-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Parallel_computing&oldid=512900467

[3] E. Bodden, M. Clasen, and J. Kneis, "Arithmetic coding revealed - a guided tour from theory to praxis," Sable Research Group, McGill University, Tech. Rep. 2007-5, May 2007. [Online]. Available: http://www.bodden.de/pubs/sable-tr-2007-5.pdf

[4] (2012, apr) CUDA C Programming Guide. Nvidia Corporation. [Online]. Available: http://developer.download.nvidia.com/compute/DevZone/docs/html/C/doc/CUDA_C_Programming_Guide.pdf

[5] (2012, aug) single instruction, multiple data (SIMD). Wikipedia. [Online]. Available: http://en.wikipedia.org/wiki/SIMD

[6] N. Corporation. (2012) Geforce gtx 480 — specifications — geforce. [Online]. Available: http://www.geforce.com/hardware/desktop-gpus/geforce-gtx-480/specifications

[7] (2005, sep) Software Optimization Guide for AMD64 Processors. Advanced Micro Devices, Inc. (AMD). [Online]. Available: http://support.amd.com/us/Processor_TechDocs/25112.PDF

[8] L. Ibanez, W. Schroeder, L. Ng, and J. Cates, *The ITK Software Guide*, 2nd ed., Kitware, Inc. ISBN 1-930934-15-7, http://www.itk.org/ItkSoftwareGuide.pdf, 2005.

[9] M. Kuiper, D. Micevski, C. Share, L. Parkinson, and P. Ward, "cuda-insight-toolkit - an itk extention to support gpu filters using cuda," 2012. [Online]. Available: http://code.google.com/p/cuda-insight-toolkit

[10] Wikipedia, "Entropy (information theory) — wikipedia, the free encyclopedia," 2012, [Online; accessed 3-September-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Entropy_(information_theory)

[11] D. Marpe, H. Schwarz, and T. Wiegand, "Context-based adaptive binary arithmetic coding in the h.264/avc video compression standard," *Circuits and Systems for Video Technology, IEEE Transactions on*, vol. 13, no. 7, pp. 620 – 636, july 2003.

[12] J. Huber, *Skriptum: Informationstheorie für Fortgeschrittene*, 2012.

[13] Wikipedia, "Shannon's source coding theorem — wikipedia, the free encyclopedia," 2012, [Online; accessed 6-September-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Shannon%27s_source_coding_theorem

[14] ISO, *Information technology – Coding of audio-visual objects – Part 10: Advanced Video Coding (ISO/IEC 14496-10:2012)*, 2012.

[15] T. Strutz, *Bilddatenkompression: Grundlagen, Codierung, Wavelets, JPEG, MPEG*. Vieweg+Teubner Verlag, 2009, no. Bd. 264. [Online]. Available: http://books.google.de/books?id=1JeQn8_RrQEC

[16] J. Supol and B. Melichar, "Arithmetic coding in parallel," *Int. J. Found. Comput. Sci.*, vol. 16, no. 6, pp. 1207–1217, 2005.

[17] J. Jiang and S. Jones, "Parallel design of arithmetic coding," *Computers and Digital Techniques, IEE Proceedings* -, vol. 141, no. 6, pp. 327 –333, nov 1994.

[18] V. Sze and A. Chandrakasan, "A highly parallel and scalable cabac decoder for next generation video coding," *Solid-State Circuits, IEEE Journal of*, vol. 47, no. 1, pp. 8 –22, jan. 2012.

[19] H. Sagan, *Space-filling curves*, ser. Universitext Series. Springer-Verlag, 1994. [Online]. Available: http://books.google.de/books?id=gUfvAAAAMAAJ

[20] Wikipedia, "Hilbert curve — wikipedia, the free encyclopedia," 2012, [Online; accessed 12-September-2012]. [Online]. Available: http://en.wikipedia.org/w/index.php?title=Hilbert_curve&oldid=511177859

[21] I. A. Advanced Micro Devices. (2012) Amd phenom™ ii processor model number and feature comparisons. [Online]. Available: http://www.amd.com/us/products/desktop/processors/phenom-ii/Pages/phenom-ii-model-number-comparison.aspx

[22] X. Li and M. T. Orchard, "Edge-directed prediction for lossless compression of natural images," *IEEE Transactions on Image Processing*, vol. 10, no. 6, p. 813–817, 2001. [Online]. Available: http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=923277

[23] MATLAB, *version 7.12.0 (R2011a)*. Natick, Massachusetts: The MathWorks Inc., 2011.

[24] R. A. U. Dipl.-Ing. Johannes Ballé. (2012) Berlios developer: Project summary - libcabac. [Online]. Available: http://developer.berlios.de/projects/libcabac

[25] X. Wu and N. Memon, "Calic-a context based adaptive lossless image codec," in *Acoustics, Speech, and Signal Processing, 1996. ICASSP-96. Conference Proceed-*

ings., 1996 IEEE International Conference on, vol. 4, may 1996, pp. 1890 –1893 vol. 4.

[26] B. Foundation. (2012) Big buck bunny. [Online]. Available: http://www.bigbuckbunny.org/

[27] Communications and R. S. Lab. (2012) Openjpeg library : an open source jpeg 2000 codec. [Online]. Available: http://www.openjpeg.org/